Taxonomy-based Annotations for Variability Management

Agustina Buccella[1] *, Maximiliano Arias[12], Matias Pol'la[12], and Alejandra Cechich[1]

[1] GIISCO Research Group
Departamento de Ingeniería de Sistemas - Facultad de Informática
Universidad Nacional del Comahue
Neuquén, Argentina
{agustina.buccella,maximiliano.arias,matias.polla,alejandra.cechich}@fi.uncoma.edu.ar
[2] Consejo Nacional de Investigaciones Científicas y Técnicas - CONICET

Abstract. Currently, variability management in software product lines requires novel mechanisms to deal with the inherent complexity of domain modeling. From this perspective, the construction of semantic artifacts, supporting the modeling and implementation of variability from users' requirements to reuse component development, gives stakeholders a framework for communication and disambiguation. Our work is based on level-domain views and driven by taxonomy-based annotations for describing variability and commonality. We illustrate the proposal through a case study in the marine ecology domain, where results showed an improvement in development time.

Keywords: Software Product Lines, Semantic Artifacts, ISO 19119 Taxonomy, Variability Management, Marine Ecology Domain

1 Introduction

Variability management is an activity dedicated to provide flexibility and a high level of reuse during the software development. Within the software product line approach, the variability activities are aimed at allowing developers to develop a set of similar applications based on a manageable range of variable functionalities according to expert users' needs.

Several variability management approaches have emerged in the last twenty years focusing on the different aspects of this activity. Proposals of techniques for variability representation by using feature models [9], UML-based representations [16, 19], variability consistency checking [4], and complete frameworks [12, 17] supporting the whole software product line development, provide novel ideas towards consolidating solutions about variability management issues. Within this

*Este trabajo está parcialmente soportado por el proyecto UNCOMA F001 "Reuso Orientado a Dominios" como parte del programa "Desarrollo de Software Basado en Reuso".

wide range of proposals, we are interested in those focused on providing semantic approaches and oriented to cover all the development process.

In this work, we propose an extension of our SPL development methodology, presented in [1], in which we applied a level-domain view for the geographic domain. Here, we took advantage of the geographic standards as starting points to extract semantics and create a service taxonomy for the domain. The service taxonomy [2] was created as a specialization of services categorized by the ISO/DIS 19119[3]. It helped stakeholders to bridge the gap among their different skills by reducing the wide spectrum of information sharing. However, in this development the variability was managed manually generating complexity problems while the SPL was maturing. Thus, in this work, we propose to follow our SPL development methodology by adding specific artifacts and activities to represent and manage the inherent complexity of the variability. The main objective is to continue taking advantage of semantic resources, such as the service taxonomy, and provide, at the same time, a variability approach specifically designed to allow a controlled design and implementation of common and variant parts of an SPL.

The paper is organized as follows. The next section presents related work in the literature highlighting the context of our approach against others. Section 3 describes the domain and organizational artifacts that have been developed as an extension of our SPL development methodology for managing variability. Section 4 describes the application of the extended methodology to a real case study in the marine ecology domain. Future work and conclusions are discussed afterwards.

2 Related Work

The number of variability management approaches has been growing at a accelerated rate for the last twenty years. Novel proposals for managing variability try to fill the gaps of several unsolved aspects of the software product line engineering. Thus, in order to clarify this wide panorama, in Table 1 we include some referenced works in the literature and classify them according to some modeling, implementation and semantic aspects. The intention of this table is to show the context of our proposal against to others in the literature.

For example, in the table we can see some of the known modeling techniques. An important set of such techniques is based on the concept of *features*, which first work was reported by Kang [8] with the FODA (Feature-oriented Domain Analysis) method. Other proposals derived from this first technique were presented in [9] with Feature-oriented Reuse Method (FORM). UML extensions represent another set of techniques in which researches propose extensions to UML artifacts for variability modeling [19, 6]. Also, other set of techniques include those using the Orthogonal Variability Model (OVM), firstly proposed in [12]. This model is represented as a separate model to avoid changing the way

[3]Geographic information. Services International Standard 19119, ISO/IEC, 2005.

Proposals	Modeling Class	Implementation Approach	Semantic Resources	SPL Phases
FORM [9]	Feature-oriented	Compositional	No	Modeling and Implementation
COVAMOF [17]	OVM-oriented	Compositional	No	Modeling and Implementation
Ziadi & Jézéquel [19]	UML-oriented	Compositional	OCL constraints	Product derivation
Haber et al. [7]	Hierarchical	Compositional	No	Variability Modeling
GEARS [11]	Feature-oriented	Annotative	No	Modeling and Implementation
La Rosa et al. [15]	Configuration models	–	Questionnaire-based	Variability Management
Siy et al. [18]	Ontology-based	–	Ontologies	Product derivation
Reinhartz et al. [14]	Textual-based	–	Ontologies	Domain Analysis
Our proposal	UML-OVM-oriented	Annotative and Compositional	Taxonomy and datasheets	Modeling and Implementation

Table 1. Classifying proposals in the literature

of representing the software artifacts. Thus, OVM is classified as an annotative approach, which annotate a base model by means of extensions [17].

In addition, for variability implementation there exists also a wide set of different proposals which apply novel mechanisms for making a simple and modifiable implementation. Main contributions can be classified in two main directions [10]: annotative and compositional approaches. Annotative approaches implement variabilities by adding annotations in the source code [11]. During product generation, source code must be removed in order to eliminate undesired variants [5]. Compositional approaches implement variabilities as distinct (physically separated) code units. During product derivation, specific components or code units are composed to each other.

Finally, we also analyzed proposals which apply semantic resources for supporting variability during the whole SPL development process [14, 15, 18, 19]. For example in [15] authors propose the use of questionnaire models as an analysis technique in order to capture system variability.

Finally, our proposal is based on the use of two semantic resources, a service taxonomy and functional datasheets [2]. They are applied and instantiated during several SPL process activities and act as a support for reusing common knowledge and experiences in the marine ecology domain and, at the same time, provide better communication channels.

3 Representing variability during SPL development

Variability management is an activity that must be taken into account during all stages of a software product line development. Specifically, during the domain engineering, each variation point must be clearly defined together with their variants within the context of a software product line. Thus, in this work, we propose a variability approach based on domain taxonomies which describe services in a particular domain together with their interactions. This approach is based on our SPL development methodology, presented in [1], in which we divide the domain engineering process into two types of analyses: *domain* and *organizational*. The domain analysis involves the management and modeling of the information included in a specific domain. It only analyzes and designs the domain in a general way. Then, the organizational analysis takes the domain analysis information to adapt it to the context of the SPL. Therefore, the domain analysis activities impact directly on the organizational analysis ones. In order to manage variability, we add specific artifacts and activities to the two analyses aforementioned.

3.1 Domain Analysis

At the top of Figure 1 we show the activities included in the domain analysis, as part of the SPL methodology [1], together with the domain artifacts which must be developed. A domain service taxonomy is the first artifact to be developed as part of the information source analysis (ISA) activity. This taxonomy classifies the possible services involved in a domain based on the daily work of different professionals. It is a semantic resource that acts as a controlled vocabulary for all participants. In addition, it must be developed by considering external resources as both de-facto and de-jure standard which contribute to the classification. Obviously, the construction of this service taxonomy can demand more or less time depending on the amount and quality of the available resources to assist this development. The second domain artifact is the definition of a reference architecture which specifies a preliminary structure for the service interactions. This structure provides the starting point for defining the place and type of dependencies of a service interaction schema. Then, the third artifact, functional datasheets, is developed by designing interactions in which the services of the taxonomy work together to fulfill domain specific functionalities within the components of the defined reference architecture.

For the specification of the functional datasheets, we propose the use of a graphical and formal template which represents the functionalities by means of a set of predefined service interactions. In our previous work [1], we had defined a textual template which allowed us to define some of these dependencies. Here, we extend it in order to improve the specification and formalize it by using UML metamodels represented by XML documents. The items included in the template contains, for each required functionality, an identification, such as a number or code, a textual name describing the main function, the domain in which this functionality is included, the list of services involved for fulfilling the functionality, a graphical notation and a set of XML files specifying the services and their interactions. For these two last items we must define the set of dependencies that allow us to represent the interactions. These dependencies involve the common interactions among common[4] and variant services.

[4]Common services are services which will be part of every product derived from the SPL

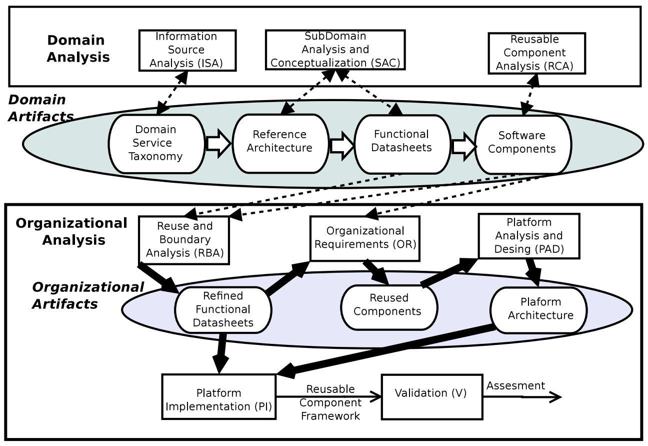
Domain Analysis
Information Source Analysis (ISA)
SubDomain Analysis and Conceptualization (SAC)
Reusable Component Analysis (RCA)
Domain Artifacts
Domain Service Taxonomy
Reference Architecture
Functional Datasheets
Software Components
Organizational Analysis
Reuse and Boundary Analysis (RBA)
Organizational Requirements (OR)
Platform Analysis and Desing (PAD)
Organizational Artifacts
Refined Functional Datasheets
Reused Components
Plaform Architecture
Platform Implementation (PI)
Reusable Component Framework
Validation (V)
Assesment

Fig. 1. Domain and organizational artifacts used to represent variability

The last artifact to be created for the domain analysis is the preliminary software component structure based on the functionalities represented on each datasheet. For each component we must indicate which functional datasheets implement, when the correspondence is 1-1; or which services, from more than one datasheet, are implementing, when the correspondence is 1-M or M-M.

Dependence Representation on Functional Datasheets

In order to represent the dependencies, we define a set of XML tags and a graphical notation, adapted from the Orthogonal Variability Model (OVM) proposed in [12]. The dependencies represented are:

– *Use dependency* (XML tag:<*Use*> $\longleftrightarrow$) specifying a dependence between common services, which are not necessarily associated with a variation point,
– *Mandatory variation point* (XML tag:<*MandatoryVP*> ———) determining the selection of a variant service when the variation point is included,
– *Optional variation point* (XML tag:<*OptionalVP*> ....) specifying that zero or more variant services, associated to the variation point, can be selected,
– *Alternative variation point* (XML tag:<*AlternativeVP*> $\wedge$) defining that only one variant service, of the set of associated variants of the variation point, must be selected (XOR relation),
– *Variant variation point* (XML tag:< *VariantVP*> $\blacktriangle$) defining that at least one variant service, of the set of associated variants of the variation point, must be selected (OR relation),
– *Requires dependency* (XML tag:*dependency:Requires = "serviceName"* $\longrightarrow$) specifying a relation between two variant services, independent from the variation points the variants are associated with, in which the selection of one variant service require the selection of the other, and

– *Excludes dependency* (XML tag:*dependency:Excludes = "serviceName"* $\not\twoheadrightarrow$) which is the opposite of the *requires dependency* specifying the exclusion of a variant when another one is selected.

As we can see, we define a simplified set of dependencies to model variability aspects. Thus, for example, we only represent requires and excludes dependencies between variants (and not between variation points [12]). This decision is based on the fact that these variability constraints are more used in practice and can be defined as simple assertions rules [17]. Then, in order to create the XML-based representation, we define three types of XML documents. The first one, named *service interactions* is generated to represent the graphical service interactions defined in a datasheet according to an UML metamodel.

The second type of XML documents is the *service information* containing the service id, the textual description, and the name of the architectural component in which it is included. Then, for each service involved in a *service interaction* XML file, a link to the *service information* XML file must be included. Finally, the third type of XML documents is the *variability constraint* which describes the variability constraints imposed to the services. Thus, for each required functionality of the domain, one template is completed by generating the functional datasheets with a set of XML files.

3.2 Organizational Analysis

The organizational analysis, as presented in [1], involves a set of five activities aimed at defining the specific organizational boundaries, and commonality and variability services. This analysis performs a refinement of the artifact developed during the domain analysis in which the context of the SPL must be specially considered. In order to illustrate the process, at the bottom of Figure 1 we show the five activities together with the organizational artifacts used and developed. As we can see, the functional datasheets and software components must be refined in order to begin enclosing the SPL to the specific applications to be derived. The XML files, with the service interactions and variability dependencies are reanalyzed resulting in a modified set of functional datasheets. Following, the refined functional datasheets and the software component structure are used to determine the final set of reusable components which conform the platform of the SPL. This structure is designed and expressed in detail, in a platform architecture. The architecture contains the information of the way each functional datasheet is implemented as software components considering also the quality requirements of the SPL. In addition, as our SPL methodology is assuming a component-based approach, in this activity we add one more item to the datasheets, a list of possible open source tools which could implement the functionality.

The next activity, platform implementation (PI), must codify the platform architecture according to the reused components and architectural requirements. As we propose a component-based approach, the first task is to analyze which components (grouping one or a set of services) will be implemented for external tools, and which of them must be fully implemented by using a determined underlying technology. In particular, the list of open source tools proposed for implementing each functional datasheet (defined during the reuse and boundary analysis (RBA) activity) must provide an initial idea of the specific software tools to be used. In addition, the quality requirements of the architecture and the component structure defined will determine the set of tools (programming languages, software platform, component models, etc.) that can implement the constraints. Finally, the last activity of the SPL methodology is focused on testing activities of the platform implementation, leaving the product development, and management of the overall product line validations for other activities involved in the product derivation phase of the SPL[5].

Dependence Representation on Software Components

In the organizational requirements (OR) activity, we define a common structure of metadata that each component should implement. In a previous work [13], we specified a variability metamodel based on an annotation system to manage some variability problems during codification. Here, we adapt this model by adding the correspondences among services included in the datasheets and components. Thus, previous to the platform implementation, each component is created with a set of annotations which follow the UML metamodel defined to represent the service interactions. For each component, we can annotate the common services involved, by means of the *Meta_use* annotation; and the variant services, by means of the group annotations of the *Meta_VariationPoint*. Each annotation has a 1-1 correspondence to the tags defined in the XML files of the functional datasheets. Finally, the annotation system allows developers to define internal variants (*Meta_DeltaInternal*) implemented as part of the same component in which the variation point is included. In our approach, we are only using the external variants in which each variant is implemented as a different component. However, the annotation system allows developers to refine the component structure to implement internal variants within the same component. Finally, *Meta_Deltas* annotation stores the information of the variability constraints within the *REQUIRES(dependency:Requires)* and *EXCLUDES(dependency:Excludes)* sets. Implementation aspects of the variability metamodel and the annotation system included on the components are described in the next section.

All these artifacts have two purposes (i) assisting in the variability management and the development of the SPL by providing a common vocabulary and interface for communication among stakeholders; and (ii) at the same time, by starting from service taxonomy, each artifact is produced by including well-formed information of the previous one in order to track the way the variability is elicited and implemented.

[5]`http://www.sei.cmu.edu/productlines/frame_report/meas_tracking.htm.`

4 Applying the model in a case study

The methodology presented here was applied to the development of new components over an SPL previously implemented. This SPL was developed by following our level-domain view within the geographic domain. There, we designed and implemented a set of open source components, each of one providing a set of services according to the domain in which they are included. The involved domains were the geographic, the oceanographic and the marine ecology. During the first stages of the SPL development, we defined a service taxonomy by specializing categorized [2] services by the Service Architecture and the ISO/DIS 19119 stds. The main objective of this taxonomy was to allow stakeholders to design functionalities of the SPL and products under a controlled range of service combinations restricting the universe of possibilities. At the same time, we defined a three level architecture with a clear and independent functionality division. It contains a *human interaction layer*, responsible for the interaction with the user; a *user processing layer*, responsible for the functionality required by the user; and a *model/information management layer*, responsible for physical data storage and data management. Each service defined in our service taxonomy, belongs to a specific architectural layer and a particular domain.

As an example of the application of the extended methodology, we illustrate a simple functionality of the platform for *querying zone attributes*. This functionality was designed as a functional datasheet modeling the service's uses and interactions within the three-layer architecture and it is part of the marine ecology domain. The left side of Figure 2 shows the graphical notation of the *querying zone attributes* datasheet and the list of the XML files created here. As we can see, the functionality is implemented by using nine services of the service taxonomy in which there are two Variant VP, PS-T2.5.1 (query the description of zones) and PS-T2.5.2 (query the code of zones). For example, the PS-T2.5.1 Variant VP can show the description attribute as a table (HI-LM1.3) or as a label (HI-LM1.20) or both. At the same time, the functionality needs the services HI-LM.5.1.2 (show zones by poligons) and MMS-FA1.1 (search zones) for searching the zones' attributes in the geographic database. Finally, two *requires* constraints are included in order to determine that if a zone code is showed by a table, the description must be also showed in the same way (HI-LM1.2 *requires* HI-LM1.3). The functionality involves eleven XML files, nine for service information, one for the service interactions, and an addition of the constraints in the variability constraint XML file.

On the right side of Figure 2 we show the software component structure for implementing the functional datasheet. As we can see, we define seven components which include a set of common and variant services. As we aforementioned, each variant service is implemented on a different component and the common services can be on the same one, as long as the services belong to the same architectural layer. Taking this into account, the PS-T2.5, PS-T2.5.1, and PS-T2.5.2 are implemented inside a same component, and the MMS-FA1.1 service is part of another one.

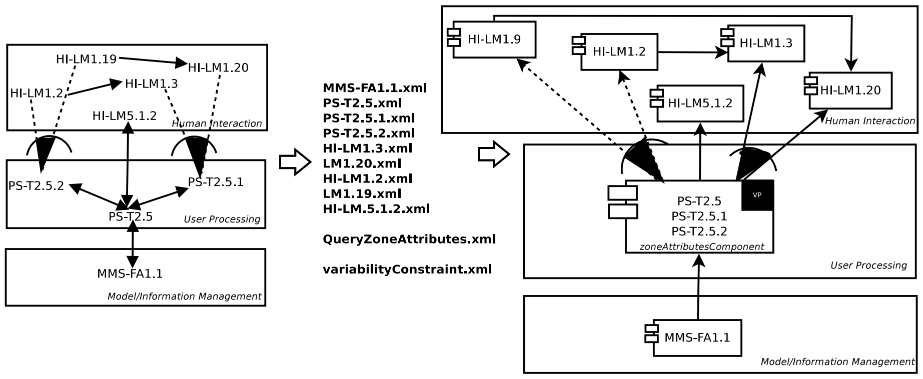

Fig. 2. Some of the software artifacts designed to represent variabilities and their dependencies

As the previous SPL was developed in Java 1.6+, we apply the particularities of this language for implementing our annotation system. Thus, we use two different programming resources. The first one is Java Annotations[6] which allows developers to add metadata to java source code. An at-sign (@) precedes an annotation keyword which is used to store information about some particular aspect. In our case, we use this resource for implementing our variability metamodel.

The second resource is to use some of the main characteristics of Delta-Oriented Programming (DOP) for SPL development. This approach allows the SPL to be represented as a core module and a set of delta modules. The first one defines a common platform for all derived products, and the seconds (the deltas) specify additions, modifications, and/or deletions of code included in the core module in order to bind a specific variability. In our SPL, the use of DOP allows us to implement internal and/or external variants according to the developer's decisions. In the example of Figure 2, we follow an external variant approach in which each variant service is implemented on a different software component.

Then, during the implementation of the components, each variation point is specified as a *dummy method* which determines the point for binding a variability. This implementation is chosen because in Java 1.6+ we can only annotate classes and methods.

In order to evaluate the extended methodology, we performed a preliminary analysis by applying two main cost factors of the SIMPLE [3] model, C_{cab} (cost to build reusable components) and C_{reuse} (cost of reusing reusable components). Both factors were adapted taking into account the previous evaluation performed in [1]. In that work, we had analyzed two aspects of the SPL, the effort required to develop the reusable components and time required for the construction of products including percentage of bugs found in reusable and specific components.

[6]Oracle Corporation. `http://docs.oracle.com/javase/tutorial/java/annotations/`

Thus, in this work, we apply these factors in order to analyze the effort (in time-terms) of writing or reusing components using the new methodology in comparison to the previous one. Specifically, for the C_{cab} evaluation, we measure, within each of the eight activities of our methodology (Figure 1), the time required for modifying, extending or using the software artifacts towards the development of new common functionalities. This analysis measures the development of five new functionalities by considering the hours required to perform each of them. Each of these functionalities has been developed by five different developers who have applied the methodology for two functionalities each one; one for the previous SPL and the other for the new one. An average of the hours required by the developers is showed in Figure 3 for the five functionalities. As we can see, we have obtained a slight improvement as compared with the previous SPL which will be increased when more functionalities are being developed.

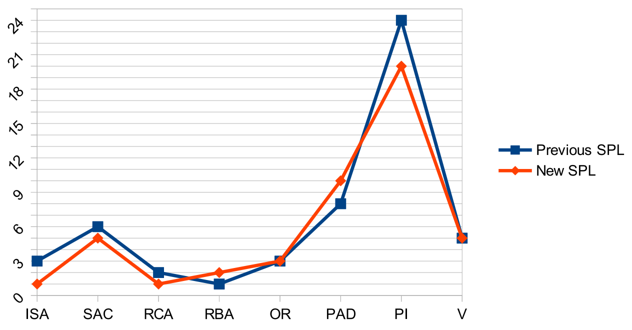

Fig. 3. Average of hours required by applying the previous and new SPL methodology

Then, for the C_{reuse} evaluation, we analyzed the cost of locating the right reusable components (looking for them into some software artifacts) and the cost of instantiating (adding, deleting, or modifying code within the components) specific variabilities included in the products. For these measures, we extended a prototype tool, implemented for assisting some tasks of the derivation process [13], in order to extract information of the software components. The information is used then for the developers for instantiating the dummy methods according to the variability designed. The use of this tool generated also improvements on the time required for the new SPL methodology.

5 Conclusion

In this article we have presented an extension of our SPL methodology [1] based on level-domain views. The proposed methodology is distinctive in three main respects. First, it proposes the use of semantic resources, such as a domain service taxonomy which stands the bases for defining functional datasheets used to specify service interactions. Second, it provides variability metamodels which are adapted depending on the activity in which the engineers or developers are working on. During the domain analysis, a variability metamodel must be used by specifying the common and variant services included in the SPL. This metamodel is translated to a set of XML files which can be machine-readable. Then, during the organizational analysis, this variability metamodel is implemented as a annotation system which will assist the process of variability. And finally, we pave the way towards automatic supporting tools for assisting the development of the software artifacts on each activity. In addition, the paper presents a real case study in which we have implemented new components by applying the extended methodology. Here, we have showed the saving obtained during the component development.

As future work, we are implementing supporting tools for assisting some artifact generations and evaluating different alternatives for improving the communication of stakeholders during the domain and product derivation phase.

Acknowledgment

We would like to thank the two organizations that are collaborating in this proyect: IBMPAS and CENPAT-CONICET. This work is partially supported by the UNComa project (04/F001) “Reuso de Software orientado a Dominios” part of the program “Desarrollo de Software Basado en Reuso”.

References

1. Buccella, A., Cechich, A., Arias, M., Pol'la, M., Doldan, S., Morsan, E.: Towards systematic software reuse of gis: Insights from a case study. Computers & Geosciences 54(0), 9 – 20 (2013), `http://www.sciencedirect.com/science/article/pii/S0098300412003913`
2. Buccella, A., Cechich, A., Pol'la, M., Arias, M., Doldan, S., Morsan, E.: Marine ecology service reuse through taxonomy-oriented SPL development. Computers & Geosciences 73(0), 108 – 121 (2014), `http://www.sciencedirect.com/science/article/pii/S0098300414002155`
3. Clements, P.C., McGregor, J., Cohen, S.: The structured intuitive model for product line economics (simple). Tech. Rep. CMU/SEI-2005-TR-003, Software Engineering Institute (2005)
4. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wäsowski, A.: Cool features and tough decisions: a comparison of variability modeling approaches. In: Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems. pp. 173–182. VaMoS '12, ACM, New York, NY, USA (2012)
5. Fenske, W., Thüm, T., Saake, G.: A taxonomy of software product line reengineering. In: Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems. pp. 4:1–4:8. VaMoS '14, ACM, New York, NY, USA (2013), `http://doi.acm.org/10.1145/2556624.2556643`

6. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (2004)
7. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.: Hierarchical variability modeling for software architectures. In: Software Product Lines - 15th International Conference. pp. 150–159 (2011)
8. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University Pittsburgh, PA. (1990)
9. Kang, K., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
10. Kästner, C., Trujillo, S., Apel, S.: Visualizing software product line variabilities in source code. In: In Proc. SPLC Workshop on Visualization in Software Product Line Engineering (ViSPLE (2008)
11. Krueger, C.W., Clements, P.: Systems and software product line engineering with biglever software gears. In: Proceedings of the 16th International Software Product Line Conference - Volume 2. pp. 256–259. SPLC '12, ACM, New York, NY, USA (2012), `http://doi.acm.org/10.1145/2364412.2364458`
12. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2005)
13. Pol'la, M., Buccella, A., Cechich, A., Arias, M.: Un modelo de metadatos para la gestión de la variabilidad en líneas de productos de software. In: Proceedings of the ASSE'14: 15th Simposio Argentino de Ingeniería de Software. Buenos Aires, Argentina (2014)
14. Reinhartz-Berger, I., Itzik, N., Wand, Y.: Analyzing variability of software product lines using semantic and ontological considerations. In: Advanced Information Systems Engineering, Lecture Notes in Computer Science, vol. 8484, pp. 150–164. Springer International Publishing (2014), `http://dx.doi.org/10.1007/978-3-319-07881-6_11`
15. Rosa, M.L., van der Aalst, W.M.P., Dumas, M., ter Hofstede, A.H.M.: Questionnaire-based variability modeling for system configuration. Software and System Modeling 8(2), 251–274 (2009)
16. Rumpe, B., Robert, F.: Variability in uml language and semantics. Software and Systems Modeling 10(4), 439–440 (2011)
17. Sinnema, M., Deelstra, S., Nijhuis, J., Bosch, J.: Covamof: A framework for modeling variability in software product families. In: Third International Conference of SPLC. LNCS, vol. 3154, pp. 197–213. Springer (2004)
18. Siy, H., Wolfson, A., Zand, M.: Ontology-based product line modeling and generation. In: Proceedings of the 2nd International Workshop on Product Line Approaches in Software Engineering. pp. 50–54. PLEASE '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/1985484.1985497`
19. Ziadi, T., Jézéquel, J.: Software product line engineering with the uml: Deriving products. In: Software Product Lines, pp. 557–588 (2006)